A Verified Compiler for an Impure Functional Language

Adam Chlipala
Harvard University
POPL 2010
What are the engineering principles that make compiler verification worth doing in the real world?

In particular, for higher-order languages, which have tricky binder issues.
From Mini-ML to Assembly

Source language

\[ e ::= \begin{array}{l}
\text{c} \mid e = e \mid x \mid e \; e \mid \text{fix } f(x). \; e \\
\text{let } x = e \text{ in } e \mid () \\
(e, e) \mid \text{fst}(e) \mid \text{snd}(e) \mid \text{inl}(e) \mid \text{inr}(e) \\
\text{case } e \text{ of } \text{inl}(x) \Rightarrow e \mid \text{inr}(x) \Rightarrow e \\
\text{ref}(e) \mid !e \mid e := e \\
\text{raise}(e) \mid e \; \text{handle } x \Rightarrow e
\end{array} \]

Target language

\[ Lvalues \; L ::= r \mid [r + n] \mid [n] \]
\[ Rvalues \; R ::= n \mid r \mid [r + n] \mid [n] \]
\[ Instructions \; I ::= L := R \mid L := R == R \mid r += n \\
\quad \mid \text{jnz } R, n \]
\[ Jumps \; J ::= \text{halt} \mid \text{fail} \mid \text{jmp } R \]

Basic blocks \[ B ::= (I*, J) \]

Programs \[ P ::= (B*, B) \]
Two Main Ideas

- It's possible to encode syntax and semantics in a way that avoids all auxiliary operations and lemmas about variables.
- Proofs about this encoding can be automated effectively enough that it is not hard to evolve a compiler and its proof over time.
Phase Structure

Conversion to higher-order syntax
CPS conversion
Closure conversion
Common subexpression elimination
Flattening
Register allocation/dead code elim.
Code generation

First-order source
Source
CPS
Closed
Three-address code
Assembly

The one and only inductive theorem about substitution
Translations with interesting binding in both source and target languages
~7000 LoC (w/ proofs)
~2000 lines of proof
Overall Compiler Correctness

\[ \text{inr}(((), 3) \]
Operational Semantics

\[(\lambda x. \, e) \, v \rightarrow [x/v]e\]

To verify \texttt{compile}, need to prove:

\[
\text{compile}([x/e_2]e_1) = [x/\text{compile}(e_2)]\text{compile}(e_1)
\]
Hiding Substitution?

$$(\lambda x. \ x) \ 1$$

App $$(\text{Lam} \ (\text{fn} \ x \Rightarrow \ x)) \ (\text{Const} \ 1)$$

$$\text{App} \ (\text{Lam} \ f) \ v \Rightarrow f(v)$$

No explicit substitution!

Adding HOAS to general-purpose proof assistants creates **unsoundness**!
Closure Semantics

Closure Heap

\( \lambda x. \ e_1 \)

\( \#n \)

\( n \)

\( [x/\#m]e_1 \)

\( \#n \)
Automating Proofs

Theorem

“By induction on....”

Case

Case

... Case

- Propositional simplification, partial evaluation, rewriting, ...
- Perform all useful **inversions** on hypotheses.
- **Choose IHes to instantiate** with unification variables.
- Finish with **higher-order logic programming** over rules of operational semantics and a few additional lemmas.
Proof Script Re-use

Lines of code added or changed to add new language features

<table>
<thead>
<tr>
<th></th>
<th>Definitions</th>
<th>Theorems &amp; Proofs</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>let</strong></td>
<td>30</td>
<td>0</td>
<td>½ hour</td>
</tr>
<tr>
<td><strong>Constants &amp; =</strong></td>
<td>150</td>
<td>10</td>
<td>½ day</td>
</tr>
<tr>
<td><strong>fix</strong></td>
<td>70</td>
<td>350</td>
<td>1 day</td>
</tr>
</tbody>
</table>

Almost all has to do with a new binding pattern, not the semantics of **fix**.
Code available in the latest **Lambda Tamer** distribution:

http://ltamer.sourceforge.net/
Backup Slides
Manipulating Binders

Which variables does the new expression mention? Are they available in scope?

```
let x = ... in
let y = ... x ... in
let u = ... x ... in
let z = ... x ... y ... in
... z ...{u...}
```

Does the new binding shadow a variable needed here?
De Bruijn Indices

Exactly which variables does this expression expect?

```
let x = ... in
let y = ... 0 ... in
let u = ... 1 ... in
let z = ... 12 ... 01 ... in
... 0 ... 1 ...
```

Did we adjust this index properly?
Higher-Order Syntax

\[
\text{let } (\ldots) \ (\lambda x. \\
\text{let } (\ldots x \ldots) \ (\lambda y. \\
\text{let } (\ldots x \ldots) \ (\lambda u. \\
\text{let } (\ldots x \ldots y \ldots) \ (\lambda z. \\
\ldots z \ldots (u \ldots ))))))
\]
Weak Higher-Order Syntax

\[
\begin{align*}
\text{let} & \quad (\ldots) \quad (\lambda x : \text{var}. \\
\text{let} & \quad (\ldots \ #x \ #y \ #u \ #z \ \ldots) \quad (\lambda y : \text{var}. \\
\text{let} & \quad (\ldots \ #x \ #y \ #u \ #z \ \ldots) \quad (\lambda u : \text{var}. \\
\text{let} & \quad (\ldots \ #z \ \ldots) \quad (\lambda z. \\
\end{align*}
\]
A piece of syntax is a first-class polymorphic function.

\[ \forall \text{var}: \]
\begin{align*}
& \text{let} \ (\ldots) \ (\lambda x : \text{var}. \ \ldots) \\
& \text{let} \ (\ldots \ #x \ \ldots) \ (\lambda y : \text{var}. \ \ldots) \\
& \text{let} \ (\ldots \ #x \ \ldots) \ (\lambda u : \text{var}. \ \ldots) \\
& \text{let} \ (\ldots \ #x \ \ldots \ #y \ \ldots) \ (\lambda z. \ \ldots \ #z \ \ldots \ #u \ \ldots)
\end{align*}